Below we summarize our observations for each algorithm:

1. **ARIMA** is a powerful model and as we saw it achieved the best result for the stock data. A challenge is that it might need careful hyperparameter tuning and a good understanding of the data.
2. **Prophet** is specifically designed for business time series prediction. It achieves very good results for the stock data but, speaking from anecdotes, it can fail spectacularly on time series datasets from other domains. In particular, this holds for time series where the notion of *calendar date* is not applicable and we cannot learn any seasonal patterns. Prophet's advantage is that it requires less hyperparameter tuning as it is specifically designed to detect patterns in business time series.
3. **LSTM-based recurrent neural networks** are probably the most powerful approach to learning from sequential data and time series are only a special case. The potential of LSTM based models is fully revealed when learning from massive datasets where we can detect complex patterns. Unlike ARIMA or Prophet, they do not rely on specific assumptions about the data such as time series stationarity or the existence of a Date field. A disadvantage is that LSTM based RNNs are difficult to interpret and it is challenging to gain intuition into their behaviour. Also, careful hyperparameter tuning is required in order to achieve good results.

ARIMA

ARIMA is a class of time series prediction models, and the name is an abbreviation for AutoRegressive Integrated Moving Average. The backbone of ARIMA is a mathematical model that represents the time series values using its past values. This model is based on two main features:

1. **Past Values**: Clearly, past behaviour is a good predictor of the future. The only question is how many past values we should use. The model uses the last p time series values as features. Here p is a hyperparameter that needs to be determined when we design the model.
2. **Past Errors:** The model can use the information on how well it has performed in the past. Thus, we add as features the most recent q errors the model made. Again, q is a hyperparameter.

An important aspect here is that the time series needs to be standardized such that the model becomes independent from seasonal or temporary trends. The formal term for this is that we want the model to be trained on a *stationary* time series. In the most intuitive sense, stationarity means that the statistical properties of a process generating a time series do not change over time. It does not mean that the series does not change over time, just that the way it changes does not itself change over time.

There are several approaches to making a time series stationary, the most popular being differencing. By replacing the n values in the series with the n-1 differences, we force the model to learn more advanced patterns. When the model predicts a new value, we simply add the last observed value to it in order to obtain a final prediction. Stationarity can be somewhat confusing if you encounter the concept for the first time, you can refer to this tutorial for more details.

Parameters

Formally, ARIMA is defined by three parameters p, d, and q that describe the three main components of the model.

- **Integrated (the I in ARIMA):** The number of differences needed to achieve stationarity is given by the parameter d. Let the original features be Y_t where t is the index in the sequence. We create a stationary time series using the following transformations for different values of d.

For d=0

In this case the series is already stationary and we have nothing to do.

For d=1

$$y_t = Y_t - Y_{t-1}$$

This is the most typical transformation.

For d=2

$$y_t = Y_t - Y_{t-1} - (Y_{t-1} - Y_{t-2}) = Y_t - 2Y_{t-1} + Y_{t-2}$$

Observe that differencing can be seen as a discrete version of differentiation. For d=1 the new features represent how the values change. While for d=2 the new features represent *the rate of the change*, just like the second derivative in calculus. The above can be generalized to d>2 as well but this is rarely used in practice.

- **AutoRegressive (AR):** The parameter p tells us how many past values to consider for the expression of the current value. Essentially, we learn a model that predicts the value at time t as:

$$\overline{y}_t = \alpha_{t-p} y_{t-p} + \alpha_{t-p+1} y_{t-p+1} + ... + \alpha_{t-1} y_{t-1}$$

- **Moving Average (MA):** How many of the forecast errors in the past should be considered. A new value is computed as:

$$\overline{y}_t = \theta_{t-q} \varepsilon_{t-q} + \theta_{t-q+1} \varepsilon_{t-q+1} + ... + \theta_{t-1} \varepsilon_{t-1}$$

The past prediction errors:

$$\varepsilon_i = y_i - \overline{y}_i$$

The combination of the three components gives the ARIMA(p, d, q) model. More precisely, we first integrate the time series, and then we add the AR and MA models and learn the corresponding coefficients.

Prophet

Prophet FB was developed by Facebook as an algorithm for the in-house prediction of time series values for different business applications. Therefore, it is specifically designed for the prediction of business time series.

It is an additive model consisting of four components:

$$y_t = g(t) + s(t) + h(t) + \varepsilon_t$$

Let us discuss the meaning of each component:

1. **g(t):** It represents the *trend* and the objective is to capture the general trend of the series. For example, the number of advertisements views on Facebook is likely to increase over time as more people join the network. But what would be the exact function of increase?
2. **s(t):** It is the *Seasonality* component. The number of advertisement views might also depend on the season. For example, in the Northern hemisphere during the summer months, people are likely to spend more time outdoors and less time in from of their computers. Such seasonal fluctuations can be very different for different business time series. The second component is thus a function that models seasonal trends.
3. **h(t):** The *Holidays* component. We use the information for holidays which have a clear impact on most business time series. Note that holidays vary between years, countries, etc. and therefore the information needs to be explicitly provided to the model.
4. The **error term** ε_t stands for random fluctuations that cannot be explained by the model. As usual, it is assumed that ε_t follows a normal distribution $N(0, \sigma^2)$ with zero mean and unknown variance σ that has to be derived from the data.

LSTM recurrent neural networks

LSTM stands for Long short-term memory. LSTM cells are used in recurrent neural networks that learn to predict the future from sequences of variable lengths. Note that recurrent neural networks work with any kind of sequential data and, unlike ARIMA and Prophet, are not restricted to time series.

The main idea behind LSTM cells is to learn the important parts of the sequence seen so far and forget the less important ones. This is achieved by the so-called gates, i.e., functions that have different learning objectives such as:

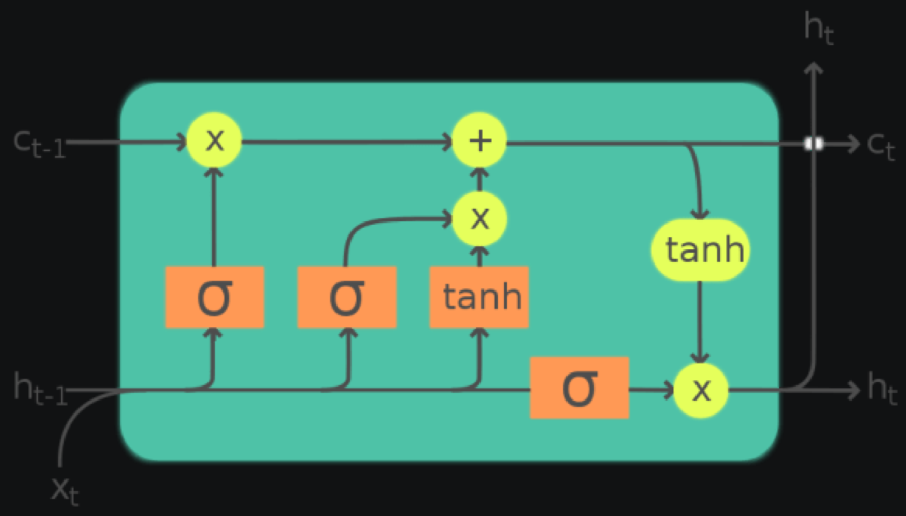
1. a compact representation of the time series seen so far
2. how to combine new input with the past representation of the series
3. what to forget about the series
4. what to output as a prediction for the next time step.

See Figure 1 and the Wikipedia article for more details.

Recurrent Neural Network Guide: a Deep Dive in RNN

Designing an optimal LSTM based model can be a difficult task that requires careful hyperparameter tuning. Here is the list of the most important parameters an LSTM based model needs to consider:

- How many LSTM cells are to use in order to represent the sequence? Note that each LSTM cell will focus on specific aspects of the time series processed so far. A few LSTM cells are unlikely to capture the structure of the sequence while too many LSTM cells might lead to overfitting.
- It is typical that first, we convert the input sequence into another sequence, i.e. the values h_t. This yields a new representation as the h_t states capture the structure of the series processed so far. But at some point, we won't need all htvalues but rather only the last h_t. This will allow us to feed the different h_t's into a fully connected layer as each h_t corresponds to the final output of an individual LSTM cell. Designing the exact architecture might require careful finetuning and many trials.



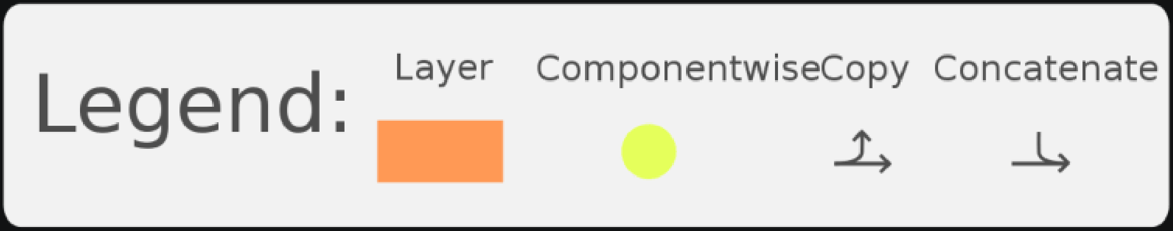


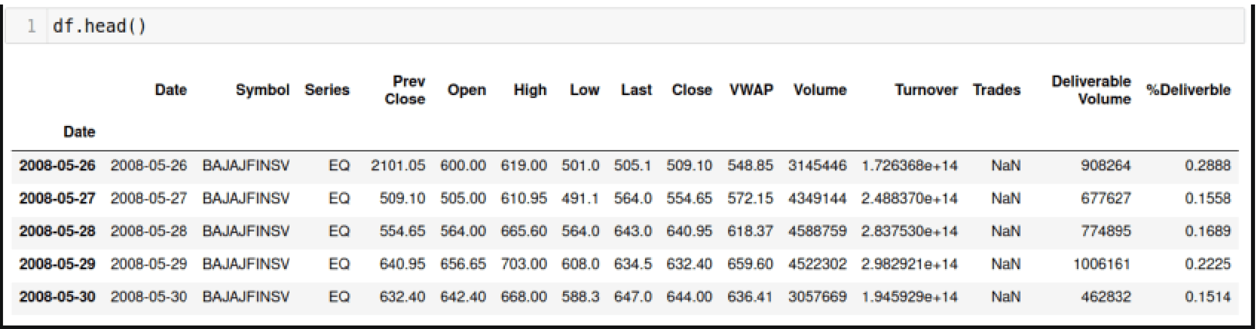
Figure 1: the structure of an LSTM cell | Source

Finally, we would like to reiterate that recurrent neural networks are a general class of methods for learning from sequential data and they can work with arbitrary sequences such as natural text or audio.

Experimental evaluation: ARIMA vs Prophet vs LSTM

Dataset

We are going to use stock exchange data for Bajaj Finserv Ltd, an Indian financial services company in order to compare the three models. The dataset spans the period from 2008 until the end of 2021. It contains the daily stock price (mean, low, and high values) as well as the total volume and the turnover of traded stocks. A subsample of the dataset is shown in Figure 2.



```
df.head()
```

	Date	Symbol	Series	Prev Close	Open	High	Low	Last	Close	VWAP	Volume	Turnover	Trades	Deliverable Volume	%Deliverble
Date															
2008-05-26	2008-05-26	BAJAJFINSV	EQ	2101.05	600.00	619.00	501.0	505.1	509.10	548.85	3145446	1.726368e+14	NaN	908264	0.2888
2008-05-27	2008-05-27	BAJAJFINSV	EQ	509.10	505.00	610.95	491.1	564.0	554.65	572.15	4349144	2.488370e+14	NaN	677627	0.1558
2008-05-28	2008-05-28	BAJAJFINSV	EQ	554.65	564.00	665.60	564.0	643.0	640.95	618.37	4588759	2.837530e+14	NaN	774895	0.1689
2008-05-29	2008-05-29	BAJAJFINSV	EQ	640.95	656.65	703.00	608.0	634.5	632.40	659.60	4522302	2.982921e+14	NaN	1006161	0.2225
2008-05-30	2008-05-30	BAJAJFINSV	EQ	632.40	642.40	668.00	588.3	647.0	644.00	636.41	3057669	1.945929e+14	NaN	462832	0.1514

Figure 2: the data used for evaluation | Source: Author

We are interested in predicting the Volume Weighted Average Price (VWAP) variable at the end of each day. A graph of the time series VWAP values is presented in Figure 3.

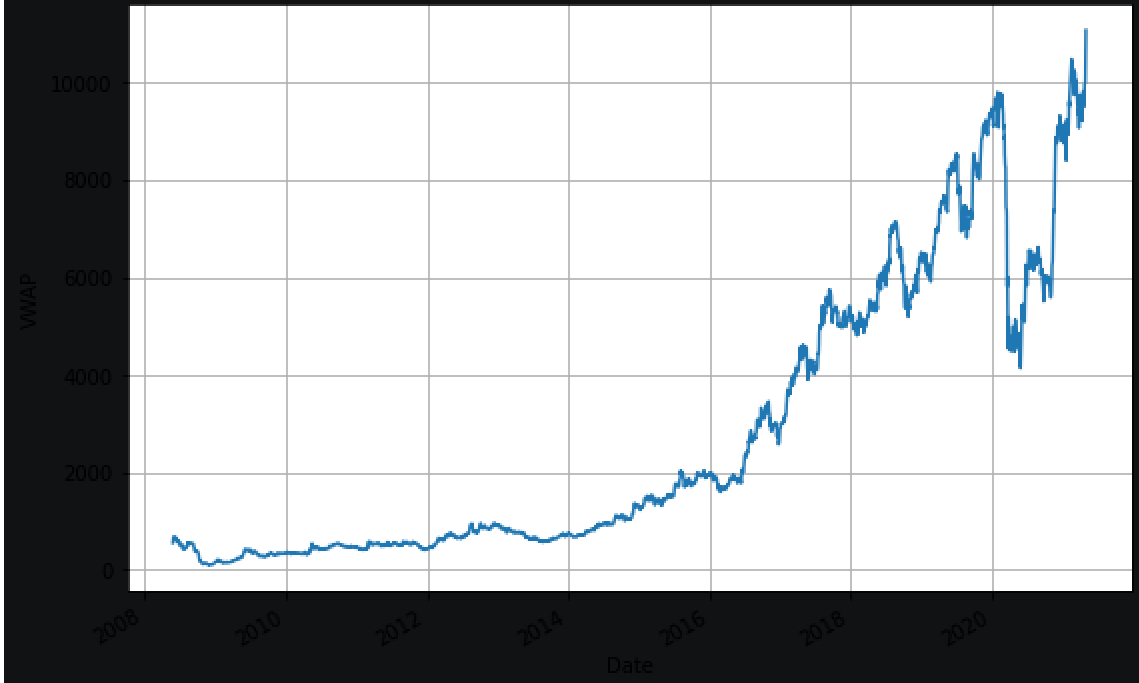


Figure 3: the daily values of the VWAP variable | Source: Author

For the evaluation, we divided the time series into a train and test time series where the training series consists of the data until the end of 2018 (see Figure 4).

Total number of observations: 3201

Training observations: 2624

Test observations: 577

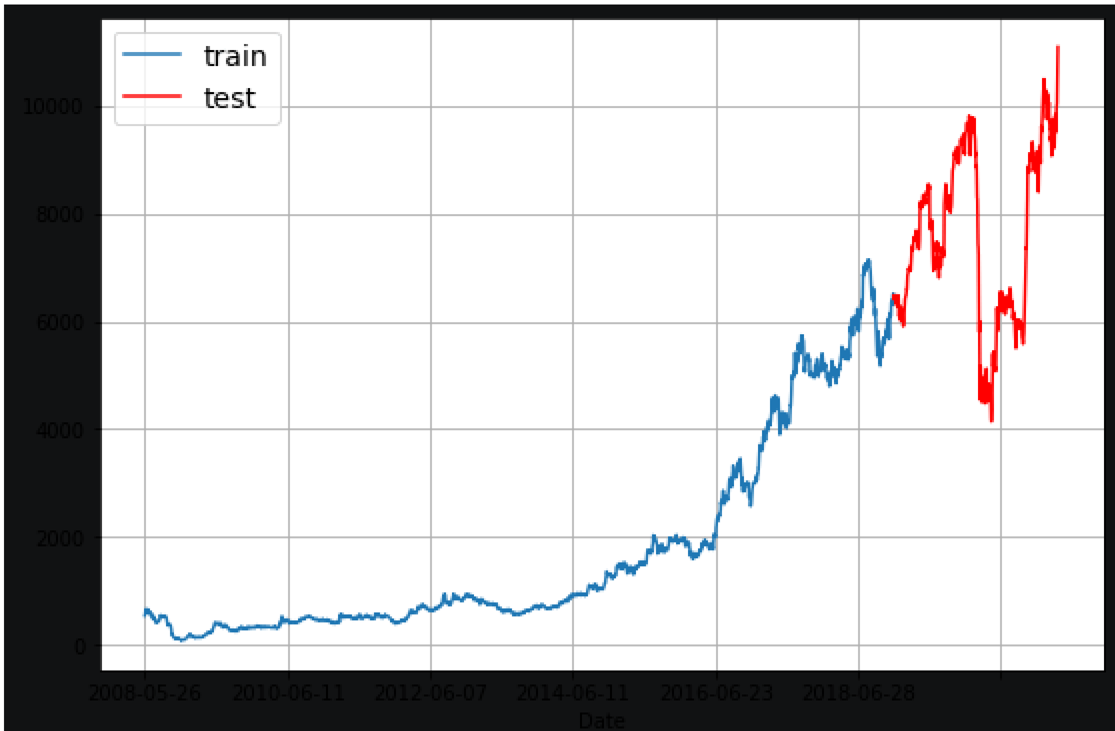


Figure 4: the train and test subsets of the VWAP time series | Source: Author

Implementation

In order to work properly, machine learning models require good data and for this, we will do a little **Feature engineering**. The objective behind feature engineering is to design more powerful models that exploit different patterns in the data. As the three models learn patterns observed in the past, we create additional features that thoroughly describe the recent trends of the stock movements.

In particular, we track the moving average for the different trade features over a period of 3, 7, and 30 days. In addition, we consider features such as the month, the week number, and the weekday. Thus, the input to our models is multidimensional. A small example of the used feature engineering looks as follows:

```
lag_features = ["High", "Low", "Volume", "Turnover", "Trades"]
df_rolled_7d = df[lag_features].rolling(window=7, min_periods=0)
df_mean_7d = df_rolled_7d.mean().shift(1).reset_index().astype(np.float32)
```

The above code excerpt shows how to add the running mean over the last week of several features describing the sales of the stock. Overall, we create a set of exogenous features:

```
exogenous_features = ["High_mean_lag3", "High_std_lag3", "Low_mean_lag3", "Low_std_lag3",
                      "Volume_mean_lag3", "Volume_std_lag3", "Turnover_mean_lag3",
                      "Turnover_std_lag3", "Trades_mean_lag3", "Trades_std_lag3",
                      "High_mean_lag7", "High_std_lag7", "Low_mean_lag7", "Low_std_lag7",
                      "Volume_mean_lag7", "Volume_std_lag7", "Turnover_mean_lag7",
                      "Turnover_std_lag7", "Trades_mean_lag7", "Trades_std_lag7",
                      "High_mean_lag30", "High_std_lag30", "Low_mean_lag30", "Low_std_lag30",
                      "Volume_mean_lag30", "Volume_std_lag30", "Turnover_mean_lag30",
                      "Turnover_std_lag30", "Trades_mean_lag30", "Trades_std_lag30",
                      "month", "week", "day", "day_of_week"]
```

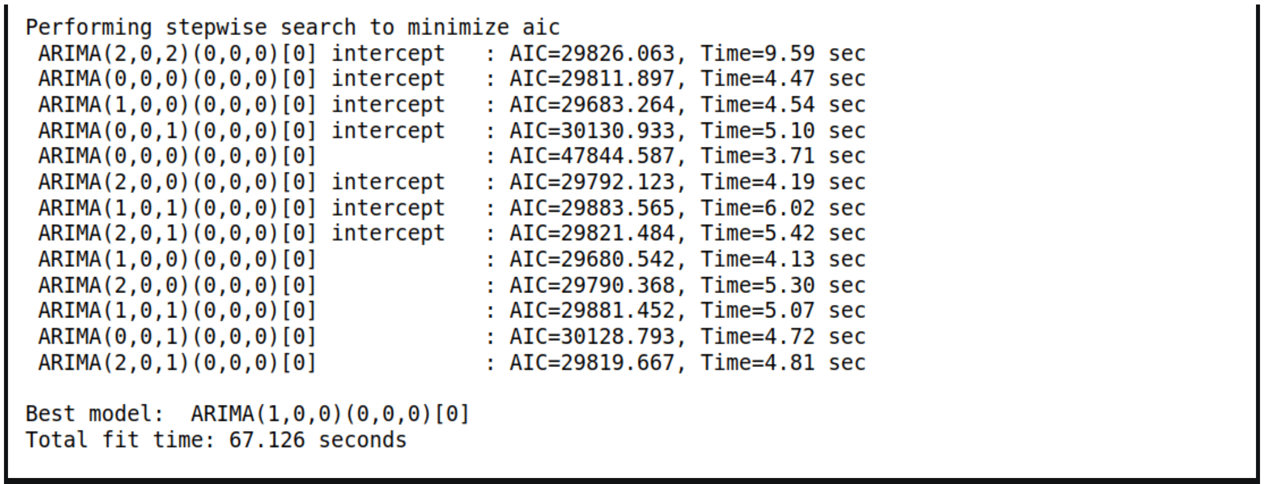
Now, let's get started with our main models:

ARIMA

We implemented the ARIMA version from the publicly available package pmdarima. The function auto_arima accepts as an additional parameter a list of *exogenous* features where we provide the features created in the feature engineering step. The main advantage of auto_arima is that it first performs several tests in order to decide if the time series is stationary or not. Also, it employs a smart grid search strategy that determines the optimal parameters for p, d, and q discussed in the previous section.

```
from pmdarima import auto_arima
model = auto_arima(
        df_train["VWAP"],
        exogenous=df_train[exogenous_features],
        trace=True,
        error_action="ignore",
        suppress_warnings=True)
```

The grid search over different values of the parameters p, d, and q is shown below. In the end, the model with the smallest AIC value is returned. (The AIC value is a measure of model complexity that simultaneously optimizes the accuracy and the complexity of a prediction model.)



```
Performing stepwise search to minimize aic
 ARIMA(2,0,2)(0,0,0)[0] intercept   : AIC=29826.063, Time=9.59 sec
 ARIMA(0,0,0)(0,0,0)[0] intercept   : AIC=29811.897, Time=4.47 sec
 ARIMA(1,0,0)(0,0,0)[0] intercept   : AIC=29683.264, Time=4.54 sec
 ARIMA(0,0,1)(0,0,0)[0] intercept   : AIC=30130.933, Time=5.10 sec
 ARIMA(0,0,0)(0,0,0)[0]             : AIC=47844.587, Time=3.71 sec
 ARIMA(2,0,0)(0,0,0)[0] intercept   : AIC=29792.123, Time=4.19 sec
 ARIMA(1,0,1)(0,0,0)[0] intercept   : AIC=29883.565, Time=6.02 sec
 ARIMA(2,0,1)(0,0,0)[0] intercept   : AIC=29821.484, Time=5.42 sec
 ARIMA(1,0,0)(0,0,0)[0]             : AIC=29680.542, Time=4.13 sec
 ARIMA(2,0,0)(0,0,0)[0]             : AIC=29790.368, Time=5.30 sec
 ARIMA(1,0,1)(0,0,0)[0]             : AIC=29881.452, Time=5.07 sec
 ARIMA(0,0,1)(0,0,0)[0]             : AIC=30128.793, Time=4.72 sec
 ARIMA(2,0,1)(0,0,0)[0]             : AIC=29819.667, Time=4.81 sec

Best model:  ARIMA(1,0,0)(0,0,0)[0]
Total fit time: 67.126 seconds
```

Predictions on the test set are then obtained by

```
forecast = model.predict(n_periods=len(df_valid),
exogenous=df_valid[exogenous_features])
```

Prophet

We use the publicly available Python implementation of Prophet. The input data must contain two specific fields:

1. **Date**: should be a valid calendar date from which the holidays can be computed
2. **Y**: the target variable we want to predict.

We instantiate the model as:

```
from prophet import Prophet
model = Prophet()
```

The features created during feature engineering have to be explicitly added to the model as follows:

```
for feature in exogenous_features:
        model.add_regressor(feature)
```

Finally, we fit the model:

```
model.fit(df_train[["Date", "VWAP"] +
exogenous_features].rename(columns={"Date": "ds", "VWAP": "y"}))
```

And the forecast for the test set is obtained as:

```
forecast = model.predict(df_test[["Date", "VWAP"] +
exogenous_features].rename(columns={"Date": "ds"}))
```

LSTM

We used the Keras implementation of LSTMs:

```
import tensorflow as tf
from keras.layers import Dropout
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from tensorflow.keras.metrics import RootMeanSquaredError, MeanAbsoluteError
from tensorflow.keras.models import Sequential
```

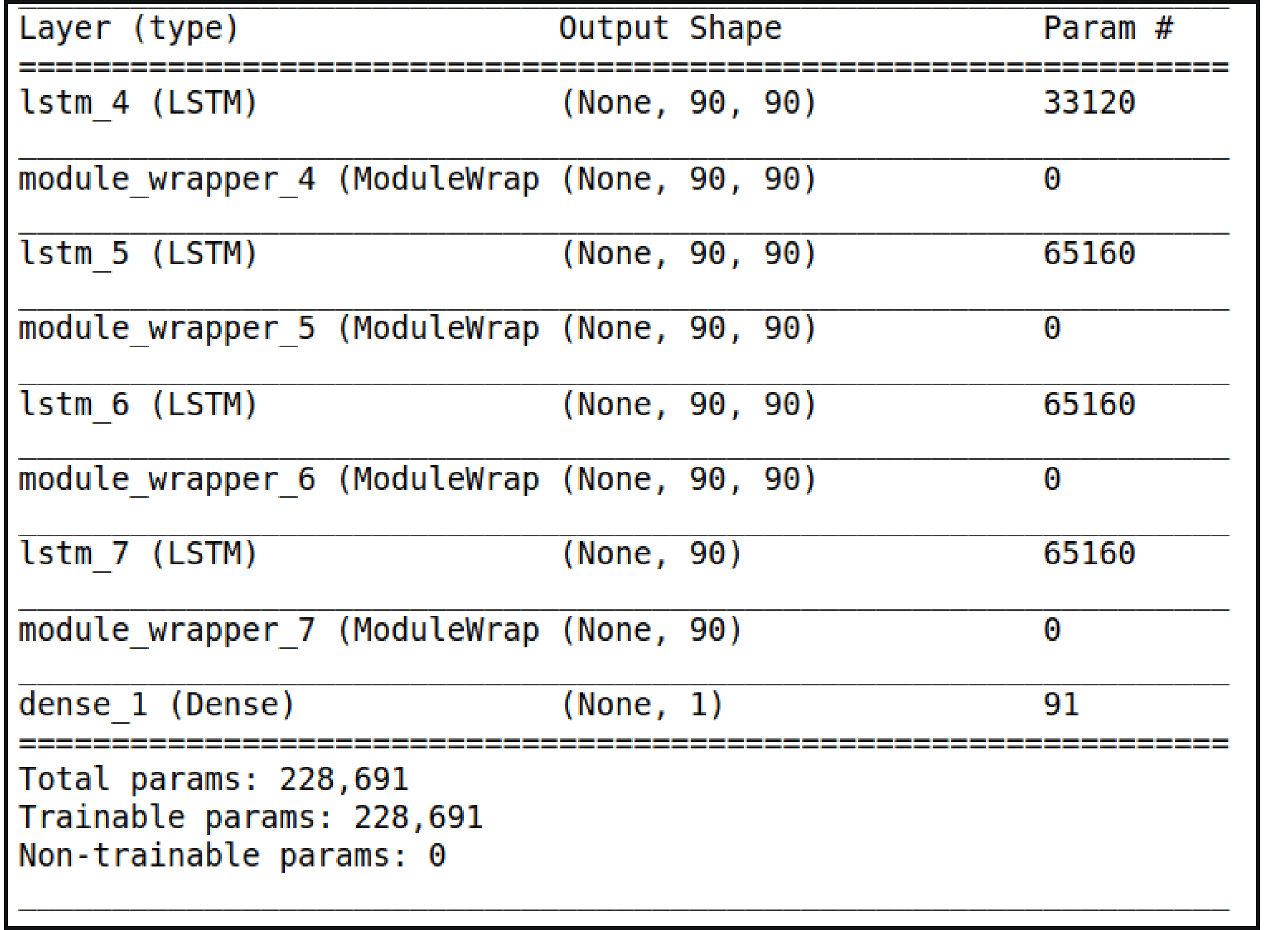
The model is defined by the following function.

```
def get_model(params, input_shape):
        model = Sequential()
        model.add(LSTM(units=params["lstm_units"], return_sequences=True,
input_shape=(input_shape, 1)))
        model.add(Dropout(rate=params["dropout"]))

        model.add(LSTM(units=params["lstm_units"], return_sequences=True))
        model.add(Dropout(rate=params["dropout"]))

        model.add(LSTM(units=params["lstm_units"], return_sequences=True))
        model.add(Dropout(rate=params["dropout"]))

        model.add(LSTM(units=params["lstm_units"], return_sequences=False))
        model.add(Dropout(rate=params["dropout"]))

        model.add(Dense(1))

        model.compile(loss=params["loss"],
                optimizer=params["optimizer"],
                metrics=[RootMeanSquaredError(), MeanAbsoluteError()])

        return model
```

Then we instantiate a model with a given set of parameters. We use the past 90 observations in the time series as a sequence for the input to the model. The other hyperparameters describe the architecture and the specific choices for training the model.

```
params = {
        "loss": "mean_squared_error",
        "optimizer": "adam",
        "dropout": 0.2,
        "lstm_units": 90,
        "epochs": 30,
        "batch_size": 128,
        "es_patience" : 10
}

model = get_model(params=params, input_shape=x_train.shape[1])
```

The above results in the following Keras model (see Figure 5):



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_4 (LSTM)                (None, 90, 90)            33120
_________________________________________________________________
module_wrapper_4 (ModuleWrap (None, 90, 90)            0
_________________________________________________________________
lstm_5 (LSTM)                (None, 90, 90)            65160
_________________________________________________________________
module_wrapper_5 (ModuleWrap (None, 90, 90)            0
_________________________________________________________________
lstm_6 (LSTM)                (None, 90, 90)            65160
_________________________________________________________________
module_wrapper_6 (ModuleWrap (None, 90, 90)            0
_________________________________________________________________
lstm_7 (LSTM)                (None, 90)                65160
_________________________________________________________________
module_wrapper_7 (ModuleWrap (None, 90)                0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 91
=================================================================
Total params: 228,691
Trainable params: 228,691
Non-trainable params: 0
_________________________________________________________________
```

Figure 5: a summary of the Keras LSTM model | Source: Author

We then create a callback to implement early stopping i.e. to stop training the model if it yields no improvement on the validation dataset for a given number of epochs (in our case 10):

```
es_callback =
tf.keras.callbacks.EarlyStopping(monitor='val_root_mean_squared_error',
                                                       mode='min',
patience=params["es_patience"])
```

The parameter *es_patience* refers to the number of epochs for early stopping.

Finally, we fit the model using the predefined parameters:

```
model.fit(
        x_train,
        y_train,
        validation_data=(x_test, y_test),
        epochs=params["epochs"],
        batch_size=params["batch_size"],
        verbose=1,
        callbacks=[neptune_callback, es_callback]
)
```

Experiment tracking and model comparison

Since in this blog post, we want to answer the simple question of which model yields the most accurate predictions for the test dataset, we will need to see how these three models fare against each other.

There are many different approaches for model comparisons such as creating tables and charts that record the evaluation of different metrics, creating graphs that plot the predicted values vs the true values on a test set, etc. However, for this exercise, we will be using **neptune.ai.**

It's a metadata store for MLOps, built for teams that run a lot of experiments. It gives you a single place to log, store, display, organize, compare, and query all your model-building metadata.

We first create a Neptune project and record the API of our account. You can check a detailed tutorial on how to do it in the Neptune documentation.

```
import neptune
```

```python
# Create a Neptune run object
run = neptune.init_run(
    project="your-workspace-name/your-project-name",
    api_token="YourNeptuneApiToken",
)
```

The variable *run* can be seen as a folder in which we can create subfolders containing different information. For example, we can create a subfolder called model and record in it the name of the model:

```python
run["model/name"] = "Arima"
```

We will compare the accuracy of these models with respect to two different metrics:

1. The root mean square error (RMSE)

$$\sqrt{\sum_{t=1}^{m} (y_t - \overline{y}_t)^2}$$

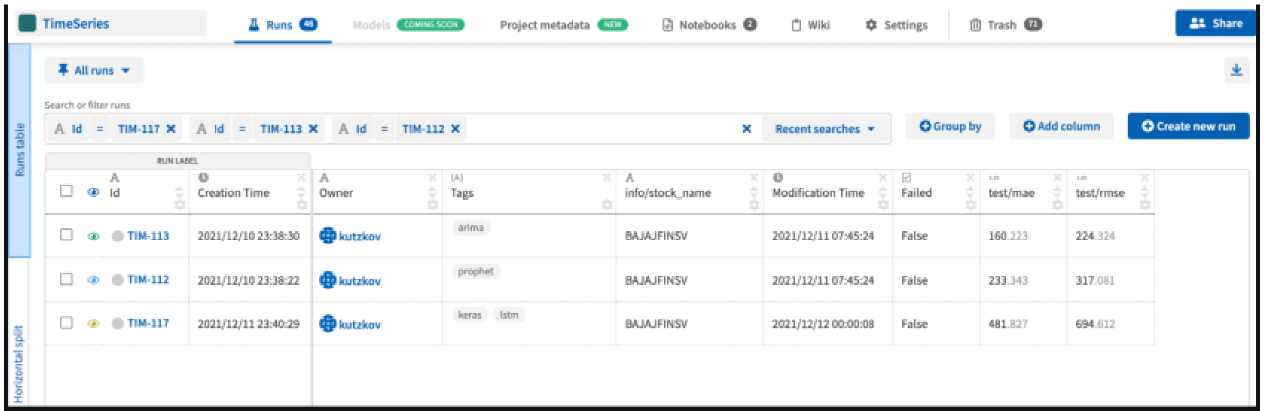
2. The mean absolute error (MAE)

$$\sum_{t=1}^{m} |y_t - \overline{y}_t|$$

Note that these values can be logged into Neptune by setting the corresponding values, for example, setting:

```python
run["test/mae"] = mae
 run["test/rmse"] = mse
```

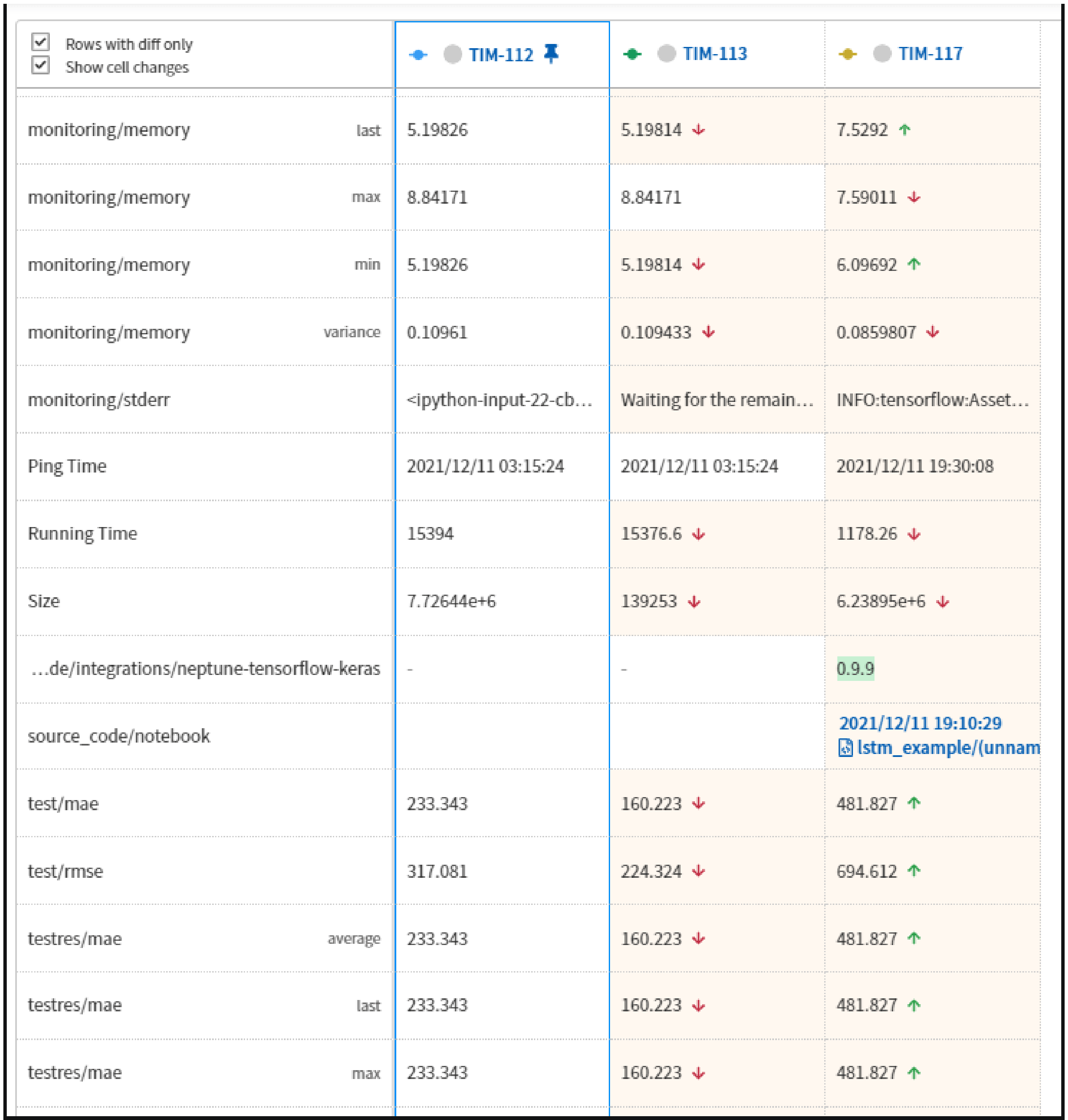
The mean square error and the mean average error for the three models can be seen next to each other in the runs table:



Id	Creation Time	Owner	Tags	info/stock_name	Modification Time	Failed	test/mae	test/rmse
TIM-113	2021/12/10 23:38:30	kutzkov	arima	BAJAJFINSV	2021/12/11 07:45:24	False	160.223	224.324
TIM-112	2021/12/10 23:38:22	kutzkov	prophet	BAJAJFINSV	2021/12/11 07:45:24	False	233.343	317.081
TIM-117	2021/12/11 23:40:29	kutzkov	keras lstm	BAJAJFINSV	2021/12/12 00:00:08	False	481.827	694.612

Figure 6. the MSE and the MAE for the three models in the Neptune web app (the tags for each project are at the top) | See in the Neptune app

The comparison of the three algorithms can be then seen side by side in Neptune, as shown in Figure 7.



☑ Rows with diff only ☑ Show cell changes		TIM-112	TIM-113	TIM-117
monitoring/memory	last	5.19826	5.19814 ↓	7.5292 ↑
monitoring/memory	max	8.84171	8.84171	7.59011 ↓
monitoring/memory	min	5.19826	5.19814 ↓	6.09692 ↑
monitoring/memory	variance	0.10961	0.109433 ↓	0.0859807 ↓
monitoring/stderr		<ipython-input-22-cb…	Waiting for the remain…	INFO:tensorflow:Asset…
Ping Time		2021/12/11 03:15:24	2021/12/11 03:15:24	2021/12/11 19:30:08
Running Time		15394	15376.6 ↓	1178.26 ↓
Size		7.72644e+6	139253 ↓	6.23895e+6 ↓
…de/integrations/neptune-tensorflow-keras		-	-	0.9.9
source_code/notebook				2021/12/11 19:10:29 lstm_example/(unnam
test/mae		233.343	160.223 ↓	481.827 ↑
test/rmse		317.081	224.324 ↓	694.612 ↑
testres/mae	average	233.343	160.223 ↓	481.827 ↑
testres/mae	last	233.343	160.223 ↓	481.827 ↑
testres/mae	max	233.343	160.223 ↓	481.827 ↑

Figure 7: The mean square error and the mean average error for the three models can be seen next to each other
(the tags for each project are at the top) | See in the Neptune app

We see that ARIMA yields the best performance, i.e., it achieves the smallest mean square error and mean absolute error on the test set. In contrast, the LSTM neural network performs the worst of the three models.

The exact predictions plotted against the true values can be seen in the following images. We observe that all three models capture the overall trend of the time series but the LSTM appears to be running behind the curve, i.e. it needs more to adjust itself to the change in trend. And Prophet appears to lose against ARIMA in the last few months of the considered test period where it underestimates the true values.

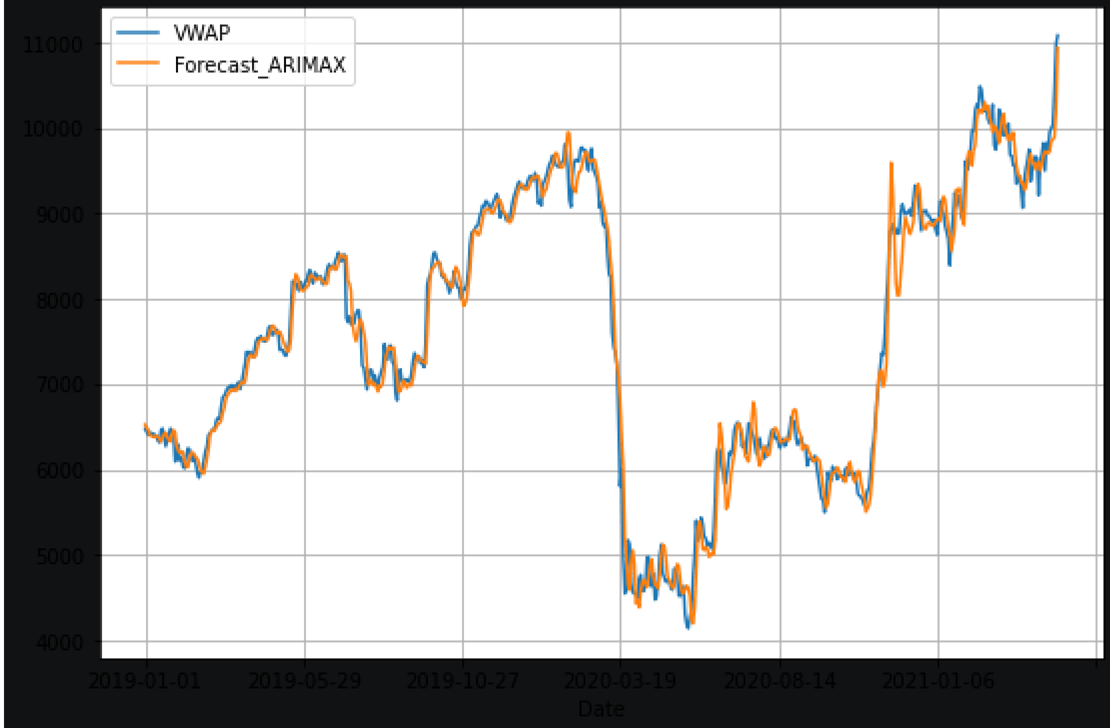


Figure 8: ARIMA predictions | Source: Author

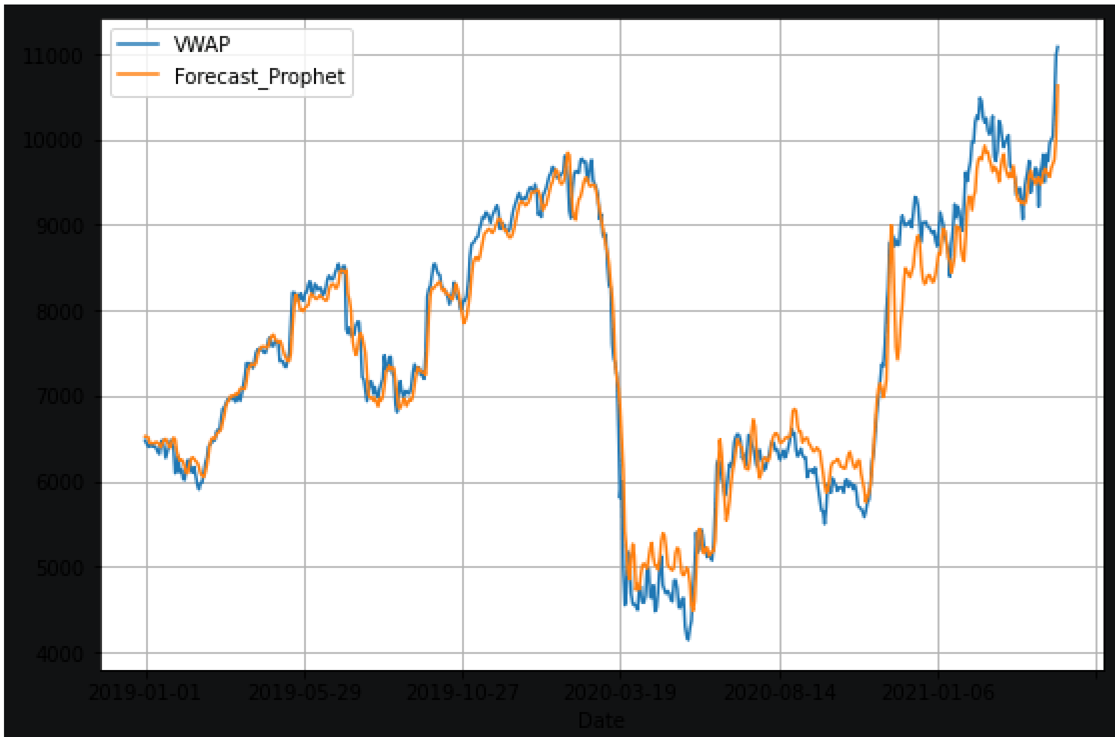


Figure 9: prophet predictions | Source: Author

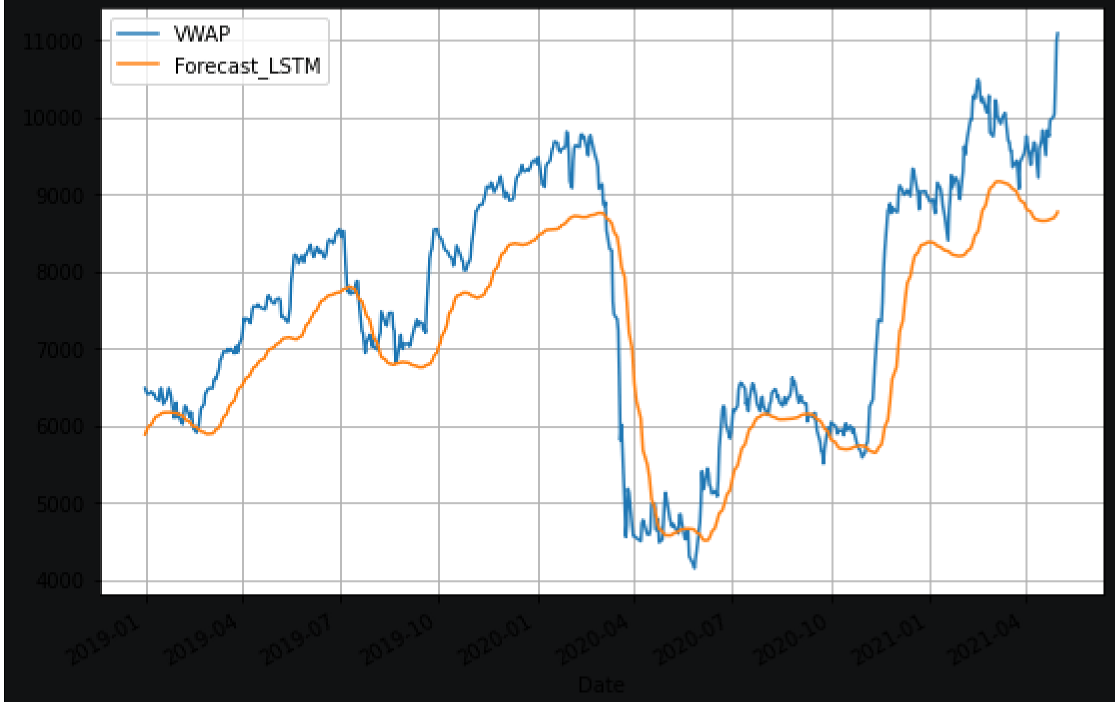


Figure 10: LSTM prediction | Source: Author

A deeper look into the performance of the models

ARIMA grid-search

When doing grid-search over different values for p, d, and q in ARIMA, we can plot the individual values for the mean squared error. The colored dots in Figure 11 show the mean square error values for different ARIMA parameters over a validation set.

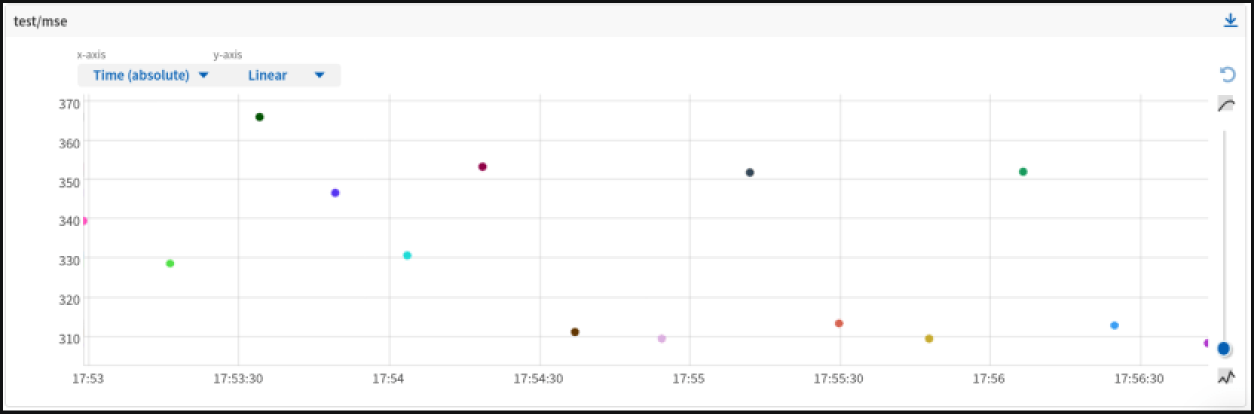


Figure 11: grid-search over the ARIMA parameters | [See in the Neptune app]

Trends in Prophet

We collect in Neptune the parameters, forecast data frames, residual diagnostic charts, and other metadata while training models with Prophet. This is achieved using a single function that captures Prophet training metadata and logs it automatically to Neptune.

In Figure 12, we show the change of the different components of the Prophet. We observe that the trend follows a linear increase while the seasonal components exhibit fluctuations.

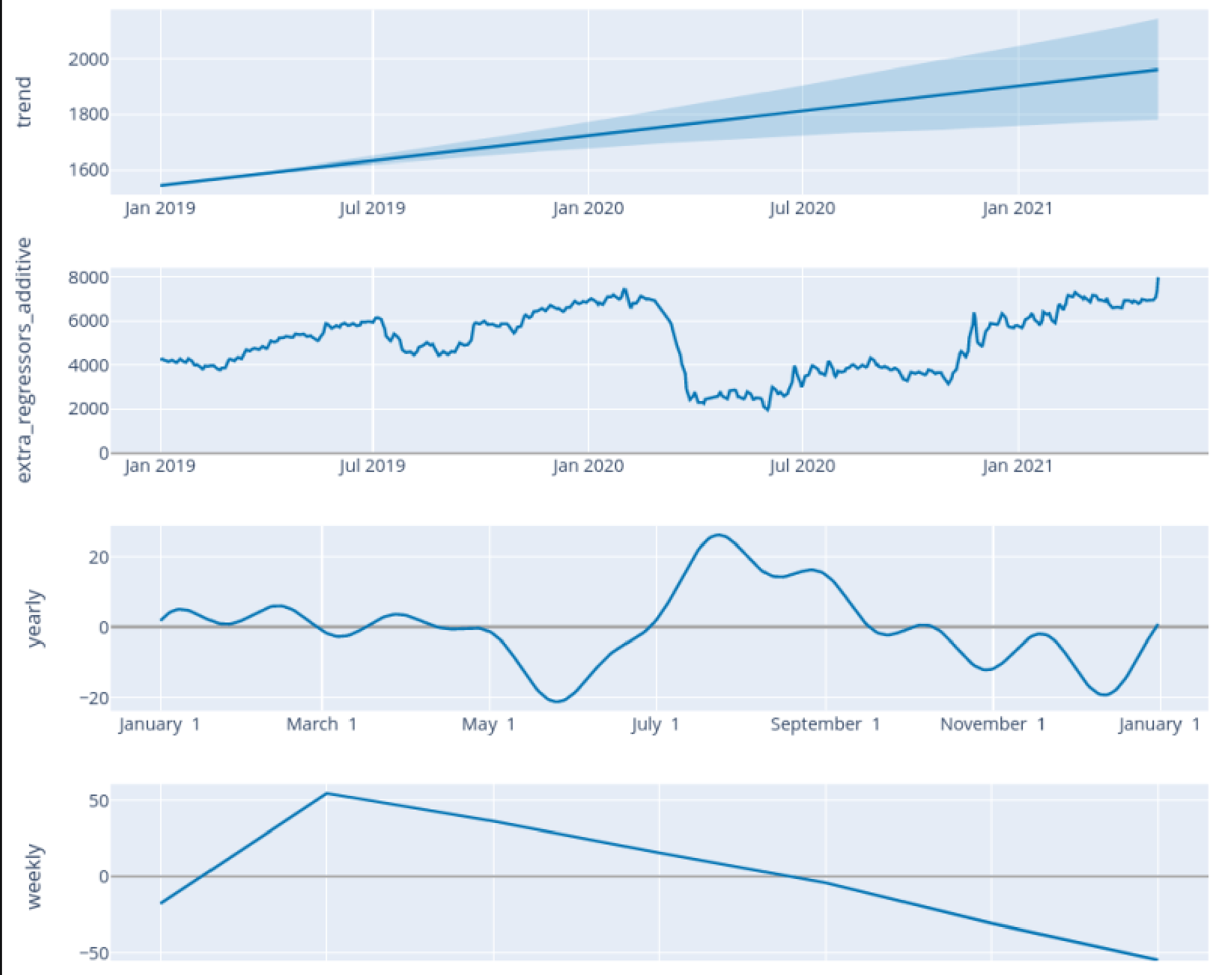
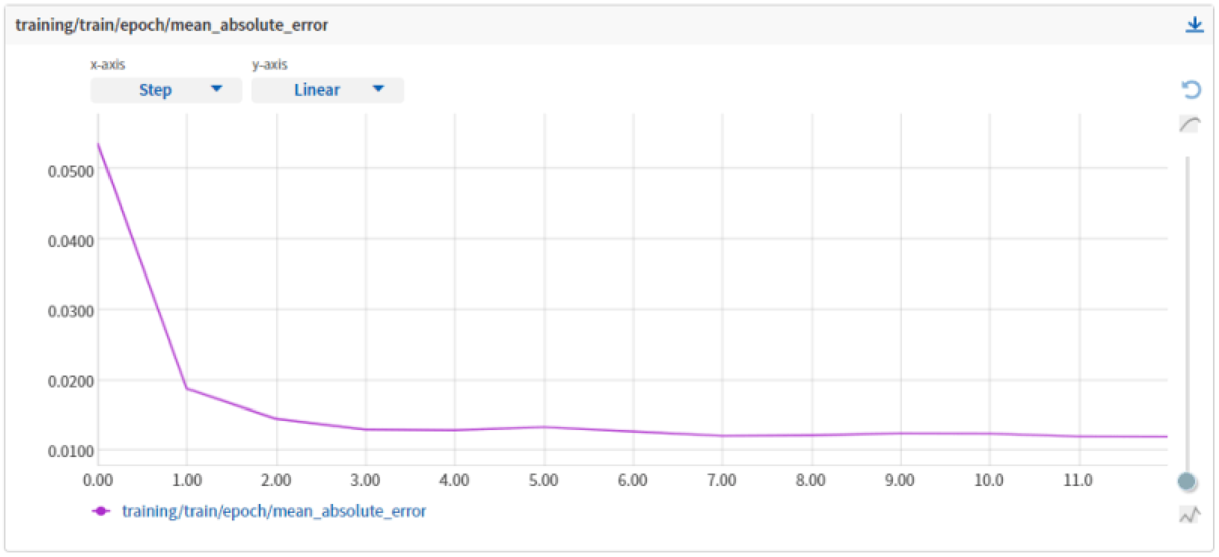


Figure 12: the change of values of the different components in the Prophet over time | Source: Author

Why did LSTM fare the worst?

We collect in Neptune the mean absolute error while training the LSTM model over several epochs. This is achieved using a Neptune callback which captures Keras training metadata and logs it automatically to Neptune. The results are shown in Figure 13.

Observe that while the error on the training dataset decreases over subsequent epochs, this is not the case for the error on the validation set which reaches its minimum in the second epoch and then fluctuates. This shows that the LSTM model is too advanced for a rather small dataset and is prone to overfitting. Despite adding regularization terms such as dropout, we can’t still avoid overfitting.



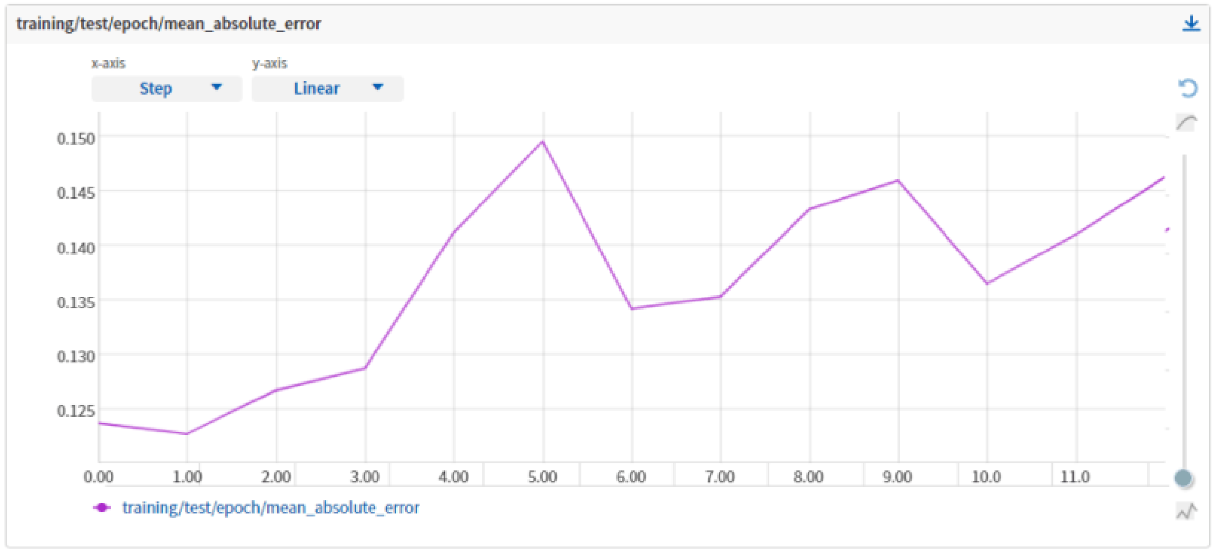


Figure 13: the evolution of train and test error over different epochs of training the LSTM model | See in the Neptune app

See how you can create dashboards in the Neptune app to analyze time-series based ML experiments.

Conclusions

In this blog post, we presented and compared three different algorithms for time series prediction. As expected, there is no clear winner and each algorithm has its own advantages and limitations. Below we summarize our observations for each algorithm:

1. **ARIMA** is a powerful model and as we saw it achieved the best result for the stock data. A challenge is that it might need careful hyperparameter tuning and a good understanding of the data.
2. **Prophet** is specifically designed for business time series prediction. It achieves very good results for the stock data but, speaking from anecdotes, it can fail spectacularly on time series datasets from other domains. In particular, this holds for time series where the notion of *calendar date* is not applicable and we cannot learn any seasonal patterns. Prophet's advantage is that it requires less hyperparameter tuning as it is specifically designed to detect patterns in business time series.
3. **LSTM-based recurrent neural networks** are probably the most powerful approach to learning from sequential data and time series are only a special case. The potential of LSTM based models is fully revealed when learning from massive datasets where we can detect complex patterns. Unlike ARIMA or Prophet, they do not rely on specific assumptions about the data such as time series stationarity or the existence of a Date field. A disadvantage is that LSTM based RNNs are difficult to interpret and it is challenging to gain intuition into their behaviour. Also, careful hyperparameter tuning is required in order to achieve good results.